University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Homework 1 is due this Friday
- Lab 1 posted on Canvas
  - Explained in recitations of this week
  - Github Classroom
- Assignment 1 will be posted this Friday

Previous lecture …

- Asymptotic analysis
- Boggle Game Problem

Boggle Game Problem (Recap)

- Words at least 3 adjacent letters long must be assembled from a 4x4 grid
- Adjacent letters are horizontally, vertically, or diagonally neighboring
- Any cube in the grid can only be used once per word

Muddiest Points

- how there were initially 15 possible ways to start the boggle solution
- Is it important to know all the members of the Big O family when O() is most widely used member?

Backtracking Framework

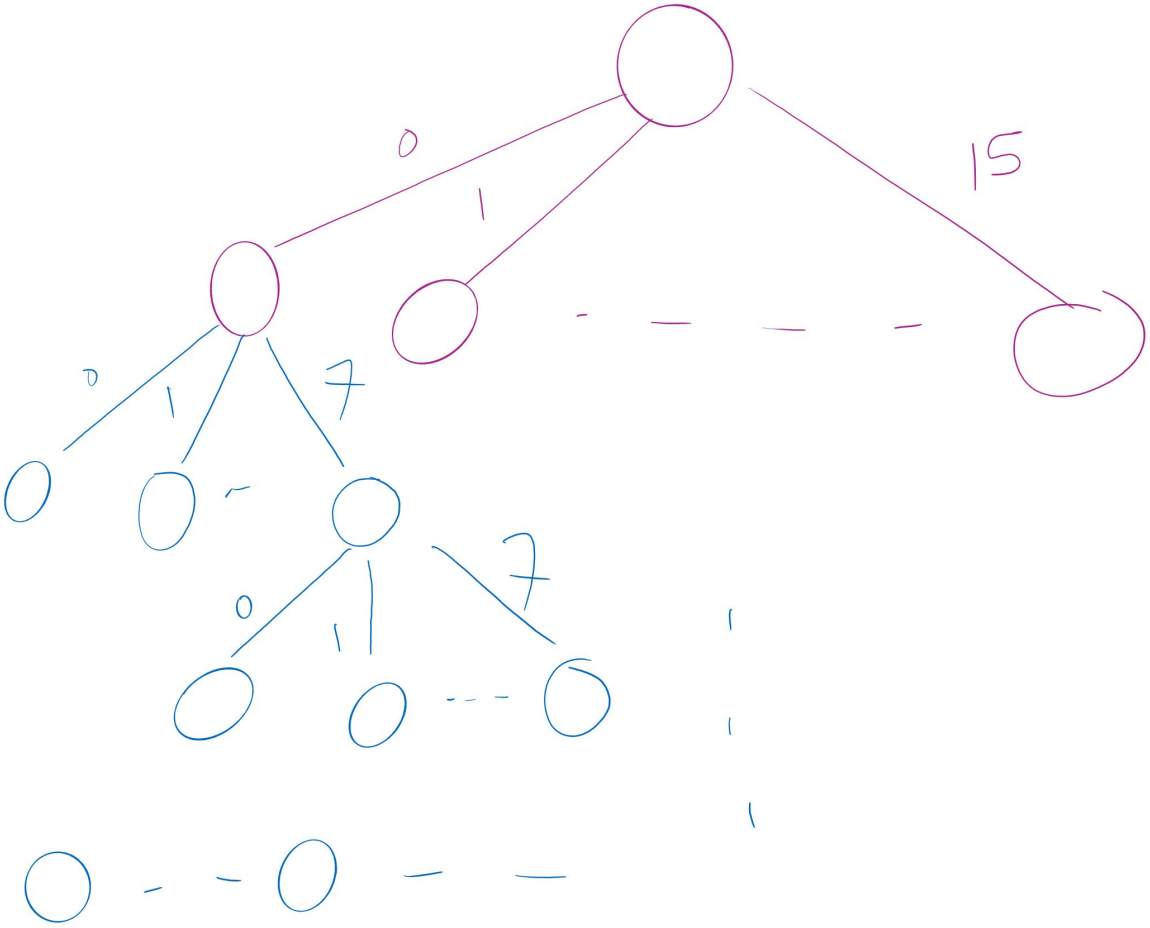
```
void traverse(current decision, partial solution) {
   for each choice at the current decision {
      if choice is valid {
         apply choice to partial solution
         if partial solution a valid solution
            report partial solution as a final solution
         if more decisions possible
            traverse(next decision, updated partial solution)
          undo changes to partial solution
      }
   }
}
```

Backtracking Algorithm for Boggle

```
void traverse(row and column of current cell, word string so far) {
   for each of the eight directions {
      if neighbor down the direction is a in the board and hasn't been used {
         append neighbor's letter to word string and mark neighbor
         as used
         if word string a word with 3+ letters
            add word string to set of solutions
         if word string is a prefix
            traverse(row and column of neighbor, word string)
          delete last letter of word string and mark neighbor as unused
      }
   }
}
```

Search Space for Boggle

- The search space can be modeled as a *tree*
- Each circle (aka node) represents one call to traverse
  - except the root node (why?)



Moving down the tree

```
void traverse(current decision, partial solution) {
   for each choice at the current decision {
      if choice is valid {
         apply choice to partial solution
         if partial solution a valid solution
            report partial solution as a final solution
         if more decisions possible
            traverse(next decision, updated partial solution)
          undo changes to partial solution
      }
   }
}
```

Backtracking (moving up the tree)

```
void traverse(current decision, partial solution) {
   for each choice at the current decision {
      if choice is valid {
         apply choice to partial solution
         if partial solution a valid solution
            report partial solution as a final solution
         if more decisions possible
            traverse(next decision, updated partial solution)
         undo changes to partial solution
      }
   }
}
```

What is the running time?

- Can't really use the frequency and cost technique because of the recursive call(s)
  - How many recursive calls are made by traverse?
- In the worst case, the backtracking algorithm must visit each node in the search space tree
- At each node, the *non-recursive* part of traverse executes
- We can use the number of nodes in the search tree as a lower bound for the worst-case runtime
  - the worst-case runtime =

    number of nodes *

    work per node (non-recursive part of traverse)

Non-recursive part of traverse

Everything but the recursive calls

```
void traverse(current decision, partial solution) {
   for each choice at the current decision {
      if choice is valid {
         apply choice to partial solution
         if partial solution a valid solution
            report partial solution as a final solution
         if more decisions possible
            traverse(next decision, updated partial solution)
          undo changes to partial solution
      }
   }
}
```

Search Space Size

- How many nodes are there?
- **Maximum number of nodes** = 1 + 16 * (…)
- = 1 + 16 * (1 + 8 + 8^2 + 8^3 + … + 8^{15})
- = 1 + 16 * $\theta(largest\ term)$
- = 1 + 16 * $\theta(8^{15})$ =
- In terms of the board size (*n*)
  - **Maximum number of nodes =** 1 + n * $\theta(8^{n-1})$
  - = $\theta(n * 8^{n-1})$ = $\theta(n * 8^n)$

Search Space Size

- In the worst case, the backtracking algorithm must visit each node in the search space
- Worst-case runtime of Backtracking for Boggle = $\Omega(n*8^n)$, where *n* is the board size (# cells)
  - if the non-recursive work is constant, that is *O(1)*, then
    - worst-case runtime = $O(n*8^n)$
  - We will see we make it constant.
- Worst-case runtime of backtracking algorithm is exponential!
  - Pruning has practical savings in runtime, but doesn't significantly reduce the runtime
    - Still exponential

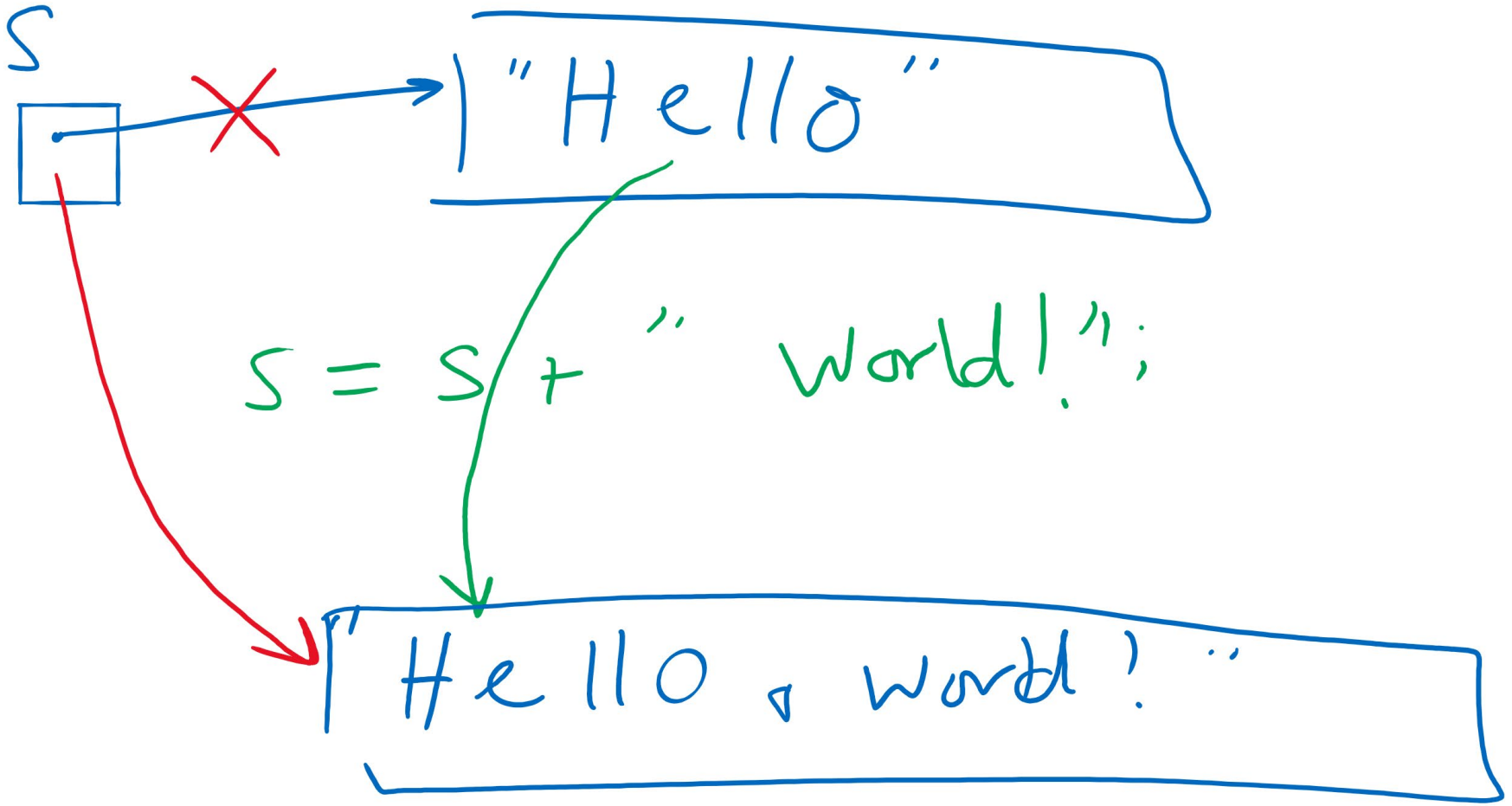
How to make the non-recursive work constant?

- How can we make the dictionary lookup (for prefixes and full words) *O(1)?*
  - Hash table? runtime?
  - Later in the course, we will see an efficient way to perform this task using a tree
- How about the time to append and delete letters from the word string?

How to make the non-recursive work constant?

- Constructing the words over the course of recursion will mean building up and tearing down strings
  - Moving down the tree adds a new character to the current word string
  - Backtracking removes the most recent character
  - Basically pushing/popping to/from a string stack
- Push/Pop stack operations are generally $\Theta(1)$
  - Unless you need to resize, but that cost can be **amortized**
- What if we use String to hold the current word string?
- Java Strings are *immutable*
  - s = new String(“Here is a basic string”);
  - s = s + “ this operation allocates and initializes all over again”;
  - Becomes essentially a $\Theta(n)$ operation
    - Where n is the length() of the string

Concatenating to String Objects



StringBuilder to the rescue

- append() and deleteCharAt() can be used to push and pop
  - Back to Θ(1)!
    - Still need to account for resizing, though…
- StringBuffer can also be used for this purpose
  - Differences?

Searching Problem

- Input:
  - a (large) dynamic set of data items in the form of
    - (key, value) pairs
  - a target *key* to search for
  - The input size (*n*) is the number of pairs
    - Key size is assumed to be constant
    - Was that case for Boggle?
- Output:
  - if *key* exists in the set: return the corresponding value
  - otherwise, return key *not found*
- What does dynamic mean?
- How would you implement “key not found”?

Let's create an Abstract Data Type!

- The Symbol Table ADT
  - A set of (key, value) pairs
  - Name dates to earlier use in Compilers
- Operations of the ST ADT
  - insert
  - search
  - delete

Symbol Table Implementations

Implementation	Runtime for Insert	Runtime for search	Runtime for delete
Unsorted Array			
Sorted Array			
Unsorted Linked List			
Sorted Linked List			
Hash Table			

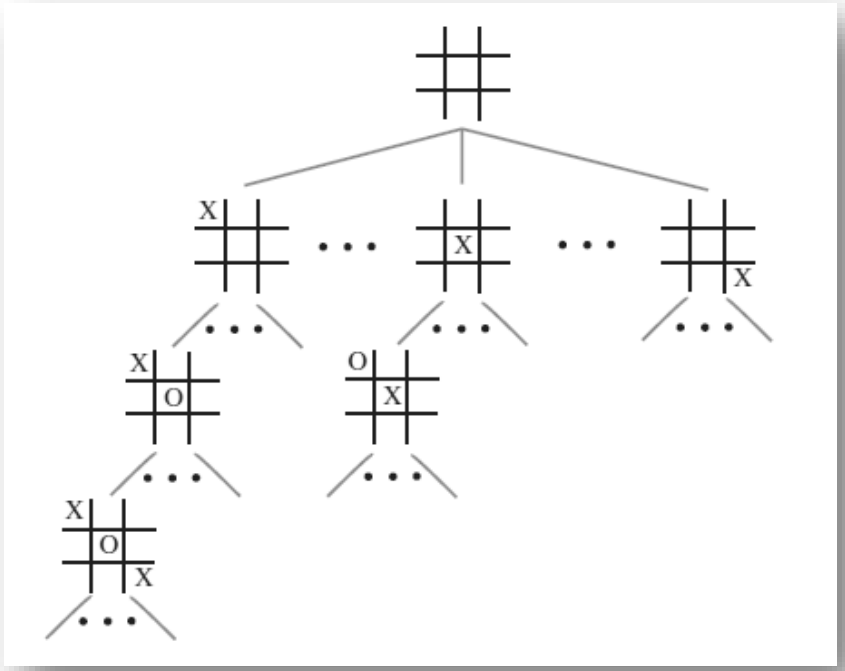
Symbol Table Implementations

Implementation	Runtime for Insert	Runtime for search	Runtime for delete
Unsorted Array	*O(1)* *amortized*	*O(n)*	*O(n)* *amortized*
Sorted Array	*O(n)*	*O(log n)*	*O(n)*
Unsorted Linked List	*O(1)*	*O(n)*	*O(n)*
Sorted Linked List	*O(n)*	*O(n)*	*O(n)*
Hash Table	*O(1) avg.*	*O(1) avg.*	*O(1) avg.*

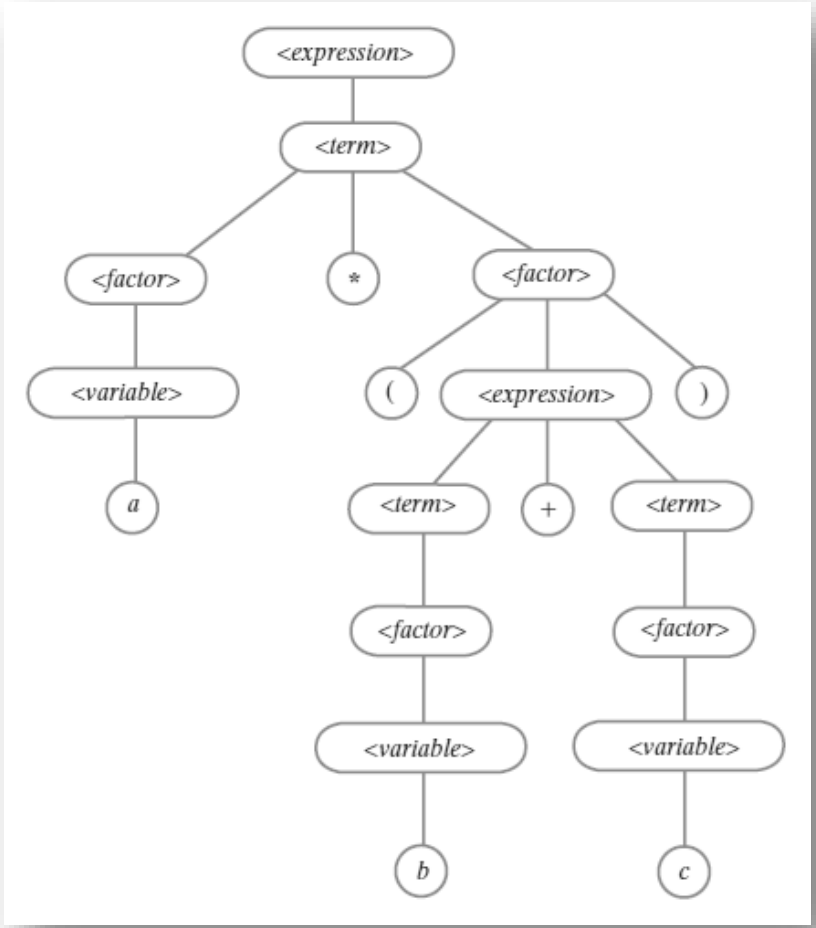
Symbol Table Implementations

- Arrays and Linked Lists are linear structures
- What if we use a non-linear data structure?
  - a Tree?

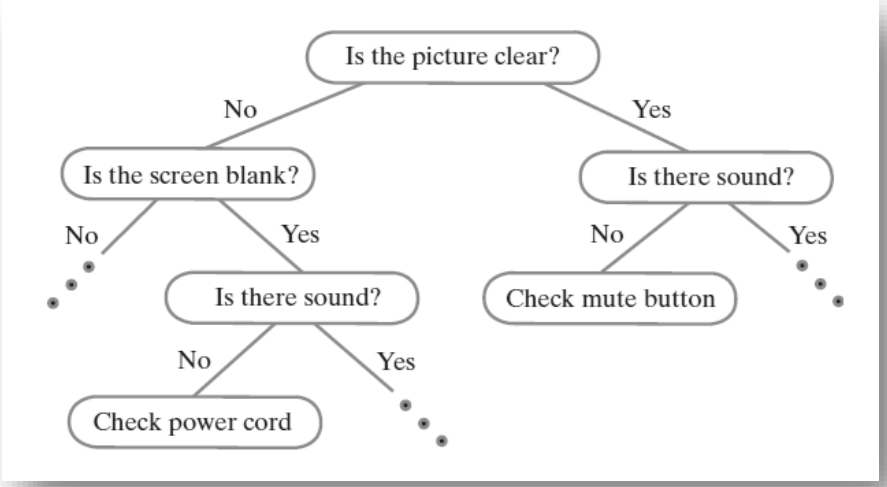
Examples of Trees



Game Tree

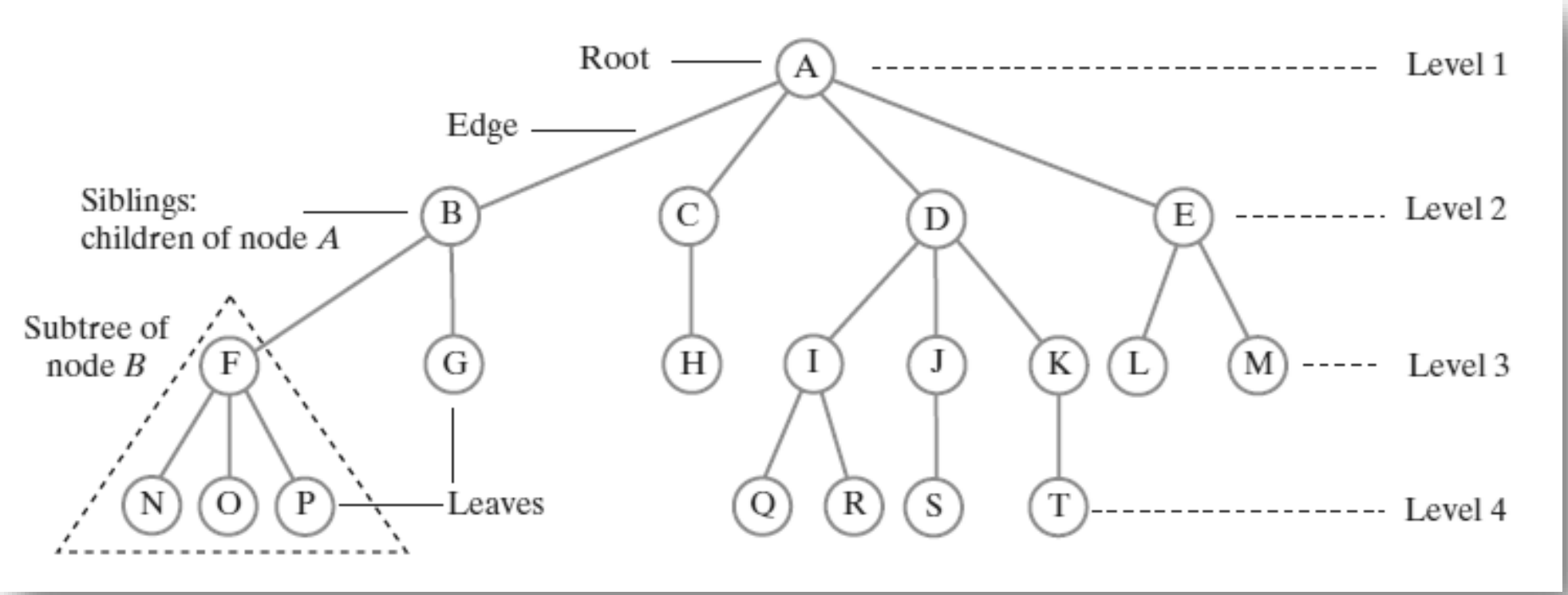


Parse Tree



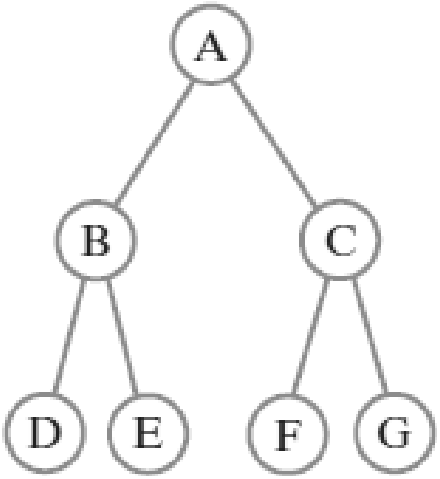
Decision Tree

Tree Terminology



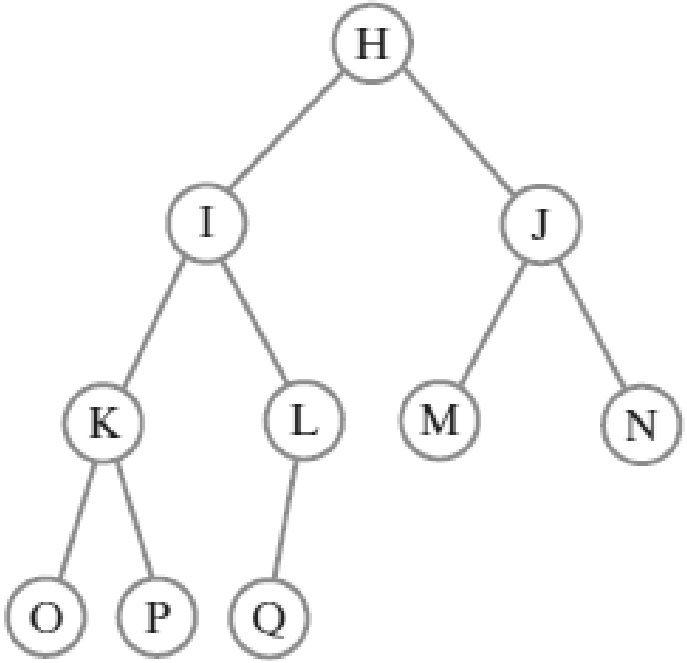
Binary Trees

(a) Full tree

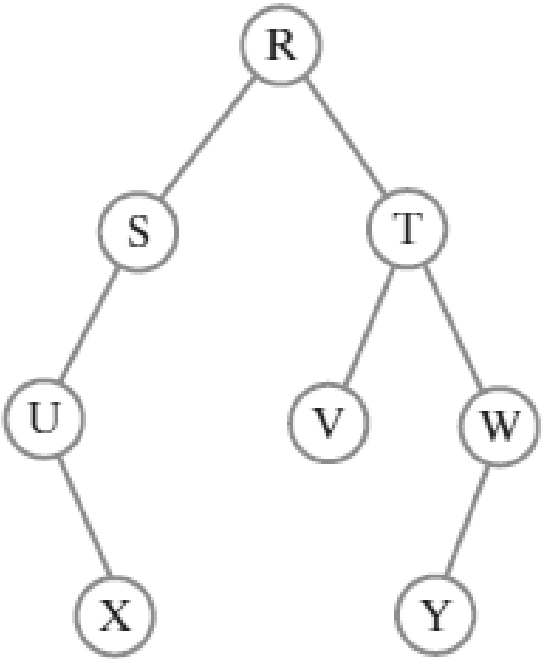


Left children: B, D, F
Right children: C, E, G

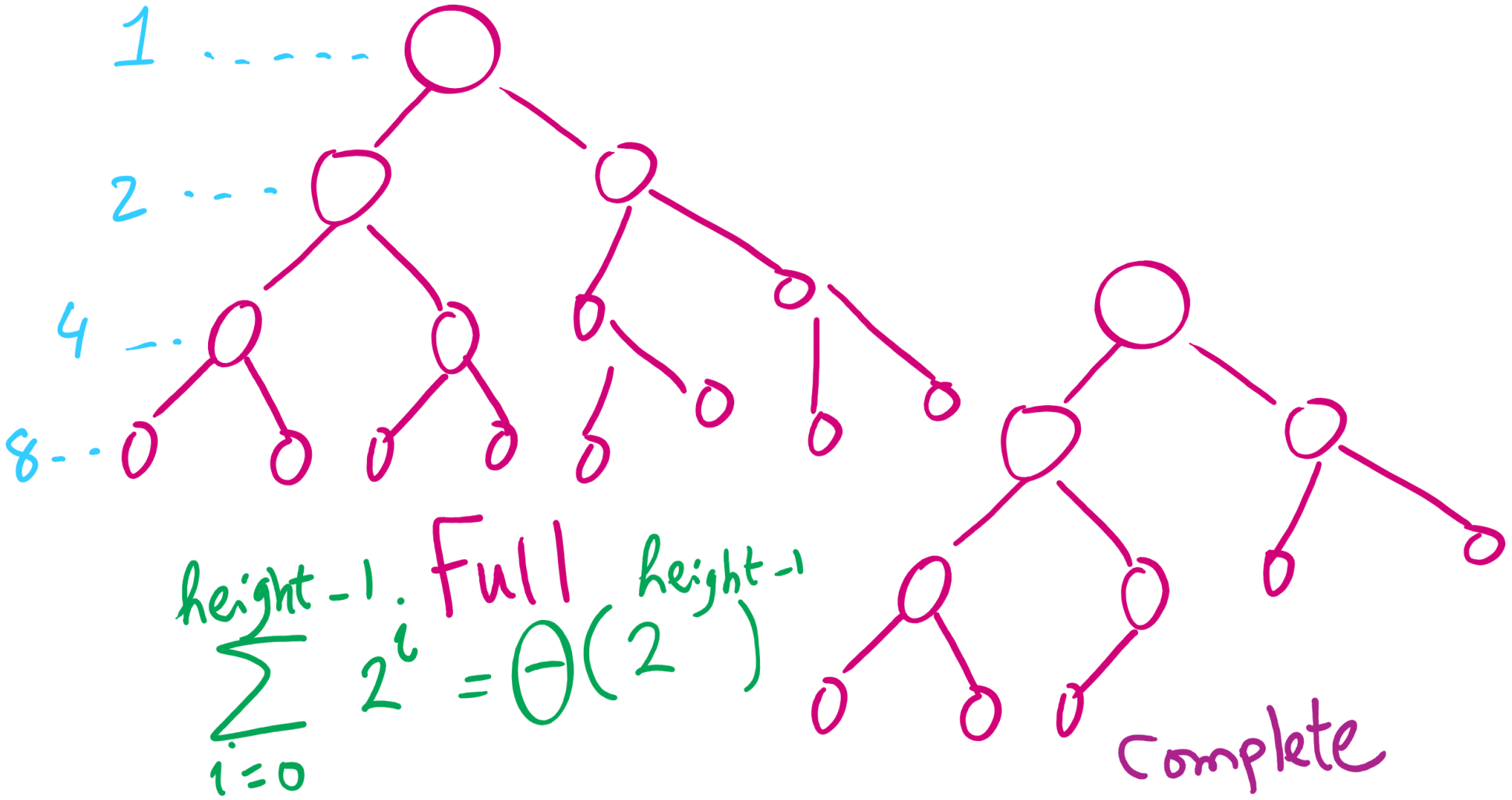
(b) Complete tree



(c) Tree that is not full and not complete



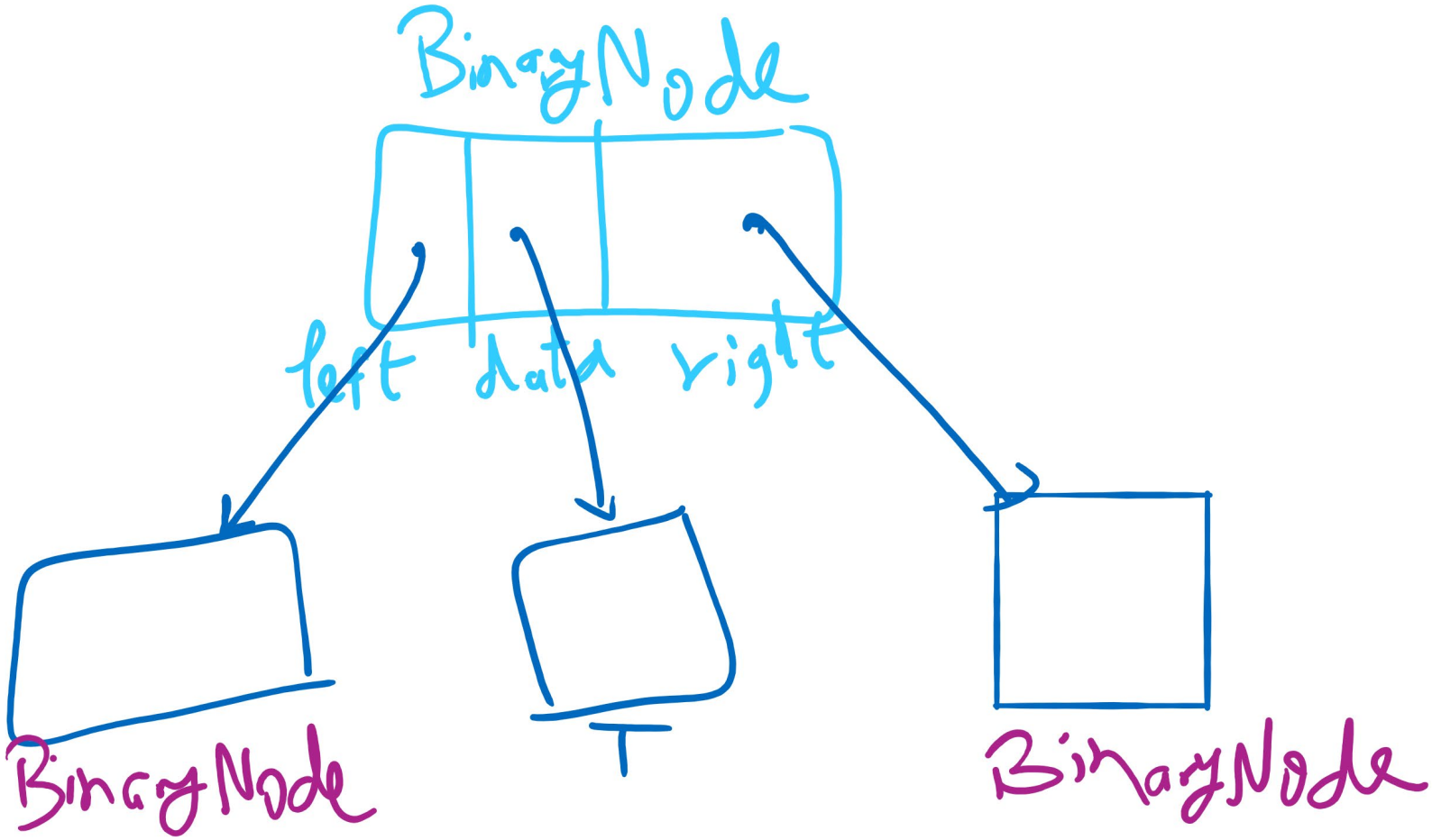
Full vs. Complete Tree



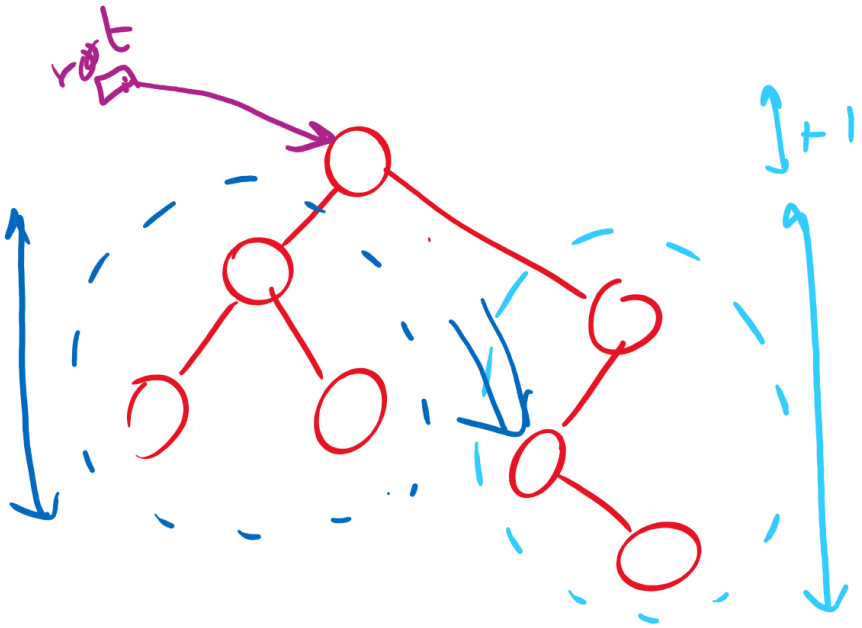
Tree Implementation: Code Walkthrough

- Available online at:
  - https://cs1501-2231.github.io/slides-handouts/CodeHandouts/TreeADT/Slides
  - The slides are under the CodeHandouts/TreeADT/slides folder in the handout repository
  - https://github.com/cs1501-2231/slides-handouts

BinaryNode



Another implementation of getHeight




```
int getHeight(BinaryNode<T> root){
    int lHeight=0;
    int rHeight=0;
    if(root.left!=null)
        lHeight=getHeight(root.left);
    if(root.right!=null)
        rHeight=getHeight(root.right);
    return Math.max(lHeight,rHeight)+1;
}
```

BinaryNode.copy